6333

TM-1042/102/00

TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of U. S. Government Contracts

Control of and Automatic Allocation for Large-Scale Cycling Systems

by

C. P. Earnest

March 15, 1963

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

CONTROL OF AND AUTOMATIC ALLOCATION FOR LARGE-SCALE CYCLING SYSTEMS

by

C. P. Earnest

1. INTRODUCTION

The computer chosen for a large program system will almost always have a primary store too small to contain the entire system at once. Rather, parts of the system are shuffled in and out as needed. Thus, part or all of the store has two dimensions effectively: space and time. Allocation of this two-dimensional store (hereinafter referred to as "working storage") presents a highly complex problem because of the interdependence of space allocation and transfer scheduling and because of the large number of independent variables involved in the allocation.

In most cases at present, any sort of reasonably sophisticated working-storage allocation is done by hand. This has the obvious disadvantages of requiring a great deal of time and being highly subject to human error. These two drawbacks, in turn, create a third: rigidity. Once an allocation is made, it is seldom changed completely; instead, slight changes are made infrequently. This creates a less than optimum allocation, and inhibits system change in general. Any change requiring an extensive reallocation is extremely difficult to implement, and might, therefore, not be made at all.

This paper presents a programmable method of storage allocation, along with a scheme for system control, suitable for large-scale systems where something

is known in advance about the sequence of program operations, and where time of system operation is an important factor. This includes, but is not limited to, most, or all, real-time systems. The allocation is static, because dynamic allocation in systems of this type usually costs more time than it saves. The primary store is assumed to be of the continuous, random-access type, such as a core memory. Throughout the present paper, the terms "primary store" and "core" are used interchangably.

The SAGE and SATIN systems have been used as models in developing the method. These systems are real-time, of very large scale, have a fairly complicated and flexible program sequence, devote about 1/3 of the computer core space to working storage, and seem to embody most of the problems common to all systems of similar type. Any methods of control and allocation that work for SAGE and SATIN should be applicable, in whole or in part, to a great many other systems. However, the study undertaken to develop the method has not been limited to these two systems; an attempt has been made to keep it as general as possible, within the framework stated in the previous paragraph.

2. OBJECTIVES

The systems used as models are run in a computer (the AN/FSQ-7), with core memory as the primary store, and drums as the major secondary store. Tapes are used for backup, simulation inputs, and system recording. The separable entities of the system are programs and tables of various lengths. These facts are included merely to establish a frame of reference for specific examples, not to limit the scope of the paper.

Certain basic assumptions must be made about the desirable and necessary characteristics of an optimum system allocation and control scheme; these assumptions follow:

1. The areas occupied by each program and its environment must be protected from the time they first appear in core until they are no longer needed.
2. Transfers should be overlapped with program operations whenever possible, in order to save time. It follows from this that:
   a. The environment of each program should already be in the primary store by the time the previous program finishes operating, insofar as possible.
   b. The transfer sequencing must be as flexible as possible. Even in a constant load situation, a given program's operating time may vary greatly.
   c. Programs and tables should be read into primary storage in the order in which they are needed, except where to do so would prevent a transfer from taking place when one is possible. For example, if program ABC operates before program DEF, DEF's environment should not be read in until all of ABC's environment is in, unless it happens that no entity for ABC can presently be read, whereas one for DEF can.
3. As few transfers as possible should be made. Transfers take time, even when overlapped with program operations.

The space available for working storage may be thought of as a limit or constraint in achieving these objectives.

Methods are presented herein which are intended to satisfy the stated objectives for working storage. A method is furnished for deciding how large working storage should be, and which entities it should contain. In addition, a method of drum allocation for systems with drums as the secondary store is presented.

3. DYNAMIC-CONTROL PROGRAM

Any allocation of core that allows for both the protection of necessary areas and the overlapping of transfers with program operations in a flexible manner is completely dependent on the scheme used to sequence programs and transfers. The allocation method to be presented assumes the existence of a central control program within the object system as the best method of handling this sequencing.

A system control program is, of course, usual; the methods it uses, in contrast, are open to wide variation. A control scheme is presented here which, it is hoped, will satisfy the requirements better than usual. The control program will be referred to as CTL; a description of it and its tables follows.

CTL will be entered at the start of each cycle (or at the beginning of the entire system run) at CTL, after the completion of each program's operation at SCX, and at intervals during the operation of all programs at SCI. The SCI entrance is used to allow CTL to initiate a new transfer if possible. It

would therefore be used only when no transfer was presently in progress, either when an I/O interrupt (indicating the completion of a previous transfer) occurs or when a programmed branch to SCI is reached.

Sequence control will be defined by two tables (SPP and SPT) of sequence parameters, which will be constructed prior to and apart from the cycle itself. SPP will contain one entry for each program that is to be operated; SPT will contain one entry for each transfer that is to be made. CTL will step through these two tables in an orderly way initiating transfers and operating programs in as efficient a sequence as possible.

To describe the operation of CTL, it will be easiest to first define several items:

PDPY - in SPT	Program dependency. Contains the number of the program entry in SPP that must have finished operating before this transfer can be initiated.
TDPY - in SPP	Transfer dependency. Contains the number of the transfer entry in SPT that must have been completed before this program can be operated.
TSTI - in SPT	Transfer started indicator. Set for each transfer in each cycle as soon as the transfer has been started (or omitted because of conditionality). All TSTI's are cleared at the beginning of the cycle.
PUNC - Single Item	Program unit count. Set to the number of the program currently operating.

TUNC - Single Item	Transfer unit count. Set to the number of the first transfer that has not yet been made.
XR5 - Single Item	Temporary TUNC. Set to the number of the next transfer that is to be made.

CTL will handle program sequencing in a straightforward manner. The programs will operate in the order they are listed in SPP. (Those whose conditionality is not met will, of course, not operate.) The only dependency on the transfer sequence is that no program will operate until its environment is in core; this, CTL will determine by checking the TDPY of the program against the current TUNC.

Transfer sequencing will be handled largely independently of program sequencing, and is somewhat more involved. CTL will attempt to initiate transfers in the order they are listed in SPT. However, if a particular transfer cannot yet be made because its PDPY is greater than or equal to the current PUNC, CTL will temporarily skip the transfer and continue stepping through SPT. XR5 will be used for stepping through SPT; TUNC is stepped only to the first transfer that has not yet been made. When each program finishes operating (SCX entrance), CTL will reset XR5 equal to TUNC, and so will again attempt to make any transfers that have been skipped. TSTI is used to prevent a transfer from being made more than once per cycle. It will probably pay to include a "skippable" indicator in SPT, because it is worthwhile skipping a transfer only if a following transfer has a smaller PDPY.

Any transfers that must be made at fixed time intervals would be made, when the time interval had elapsed, in preference to (before) any other transfer except a critical WRT.* A critical WRT transfer is defined as one that must be made as soon as possible after a given program has finished operating (weapons output tables, for instance).

A few words on the structure of the SPT table are in order here. The RDS transfers will be listed in the order that the entities are needed by the programs; the desirability of this has been previously explained (in Objectives). The PDPY of each will be set to as small a value as possible, based on the core allocation. Where two RDS are needed at the same time, the one with the smaller PDPY will be listed first. The critical WRT transfers will be placed following the last RDS for the program which makes up the table for the WRT transfer, and will, of course, have a PDPY of that same program. The non-critical WRT transfers will also have a PDPY of the program which makes up the tables. They are expected to be in core for a certain amount of time (set to as large a value as possible, based on the core allocation) after that program finishes operating, and will be listed following the last RDS for the last program that operates during that expected time, after any critical WRT transfers at the same point. In other words, if it is possible, a backlog of

---

*The assumption is made here that the area for these transfers is reserved permanently. If this is impossible, it would be necessary to restrict the time during which the transfers could be made or to dynamically find space for them.

environment for several programs is built up, to allow the WRT to be made without delaying the system.

It must be kept in mind that the order of listing in SPT is not necessarily the order of actual transfer; in fact, it is probably very rarely so. A few facts are pointed out which may serve to clarify the working of the control method and demonstrate its extreme flexibility:

1. A critical WRT transfer will almost certainly be skipped at least once. Transfers will continue to be made until the program making up the critical WRT table has finished operating. The next transfer to be started will then be the critical WRT.
2. If one program sets the conditionality for another, the second program's environment obviously cannot be read in until the first program has finished operating. However, there is no reason that environment for future programs cannot be read in if there is room for it in core. It should perhaps be pointed out here that each entity is allotted a space in core for a certain number of program operations, and that this space is thus available for it whether or not any of the surrounding environment is in core. (The same area, however, may be allotted to more than one entity, if they are mutually exclusive.)
3. In some situations it may be necessary to read in a table just before a program operates, not sooner. (The radar input tables in SAGE are presently handled this way, for instance.) Again, there is no reason to stop and wait to make these transfers; following transfers can be made until the program just preceding the one in question has finished

operating whereupon the next transfer to be initiated will be the skipped one. If there is not time to make any following transfers, the critical RDS transfer will not be skipped, and the system will not be delayed.

Note that the construction of the two sequence parameter tables is a trivial task, once the core allocation has been accomplished.

It seems to the author that the outlined method of program control almost completely satisfies the basic properties assumed to be desirable. It allows for great flexibility and perfect control at the same time. It should be pointed out that this control method can be used even if the method of allocation to be presented is not. It should also be pointed out that, with a few minor modifications, the method can be quite readily used on computers with more than one I/O channel or more than one compute module.

4. WORKING-STORAGE ALLOCATION

Several definitions are needed for the discussion:

ADVANCE AREA (A area) - That area of storage that is reserved for an entity (which must be read in) before it is actually needed by a program (i.e., it must be in core before the program can operate). This area is, of course, equal in length to the entity in question.

HOLDOVER AREA (H area) - That area of storage that is reserved for an entity (which must be written out) after it is actually needed by a program.

COMMUNICATION AREA (C area) - That area of storage that is reserved for a (nontemporary) entity after it is needed by one program, and before it is needed by another.

MUTUALLY EXCLUSIVE ENTITIES - Two entities are mutually exclusive if they can never appear in primary storage at the same time, even though they appear in the environment of the same program.

It is clear from the description of the system control program that the amount of flexibility realizable in the transfer sequencing is a function of the working-storage allocation. The sequencing gains flexibility as the advance and holdover areas increase in width.

The problem of optimum static allocation of working storage is thus one of maximizing these areas, subject to the constraint that all entities must fit in the available space without overlap.* When the problem is stated this way, it sounds as though one of the mathematical programming techniques such as linear or dynamic programming might provide a solution. However, a moment's reflection will show that the function determining whether a given set of entities will fit is not linear, or even continuous; further, adding or deleting one entity may change the value of the function completely. The function is in fact combinatorial. Therefore one method of solution is to try all the possible combinations of locations of entities for all the possible combinations of advance, holdover, and communication areas to find the optimum solution. This is clearly impractical.

*Overlap is of course permissible for, but only for, mutually exclusive entities.

It is possible, however, to split the problem into two parts: that of setting advance, holdover, and communication areas optimally, and that of fitting the resultant fixed-dimension entities into the available space. Because of the constraints acting on each of these parts of the problem, it is necessary to allow for trial of only a small subset of the total possible number of combinations.

The splitting of the problem has several important advantages in addition to that of making a practical solution possible. The setting of A, H, and C areas is highly system dependent; moreover, it is quite difficult to define and obtain a rigorous solution for this part of the problem. The problem of fitting fixed-dimension entities into the available space, on the other hand, is almost completely system independent; furthermore, its solution can be much more easily defined and the problem solved much more rigorously. (This is not meant to imply that any great degree of rigor, in the mathematical sense, has been achieved in the present paper. The orientation herein is completely toward a workable, more than an elegant, method.)

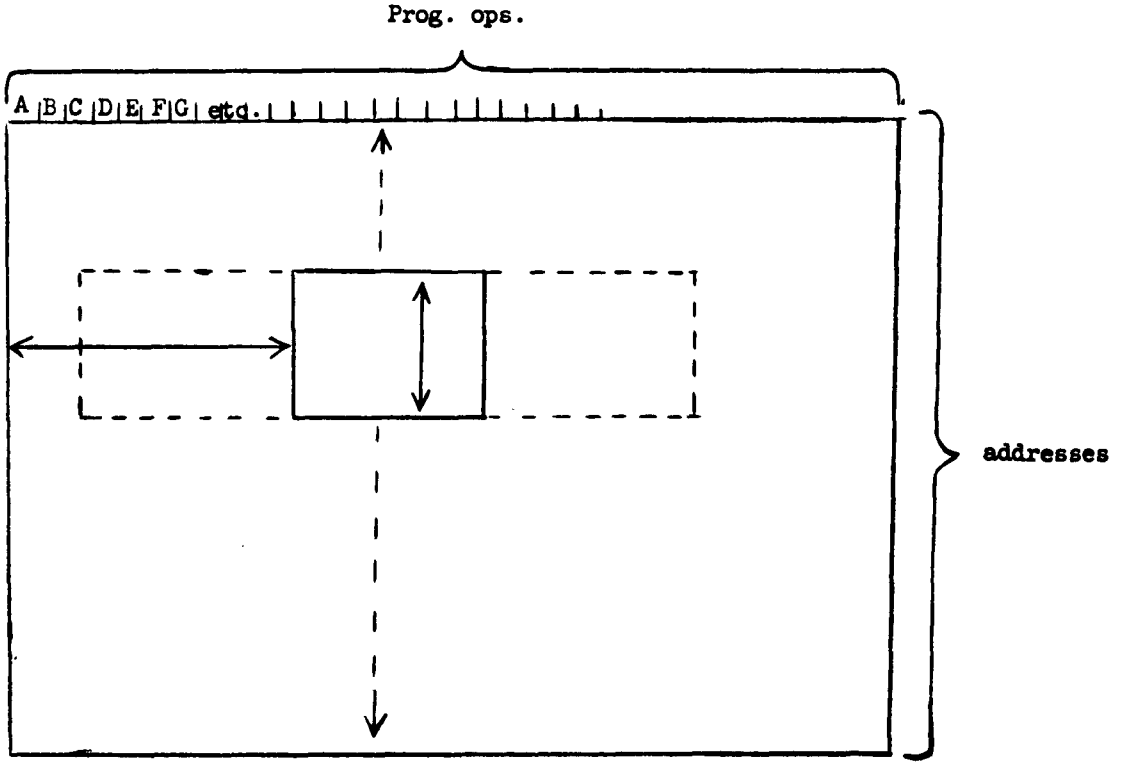
For the above reasons, then, the problem has been split into two parts, which will be presented separately.

4.1 SETTING ADVANCE, HOLDOVER, AND COMMUNICATION AREAS

Working storage can be thought of as a rectangle representing a single complete system cycle, with time as the horizontal dimension and space as the vertical dimension. The horizontal measurement is in units of program operations, with each column representing a single program; space is measured

in machine registers or words. The programs and tables (entities) of the system can be thought of as individual blocks of fixed vertical dimension (length) and horizontal position (sequence), with the horizontal dimension (A, H, and C areas) and vertical position (address) variable as shown in Figure 1.

Working Core Rectangle With Entity



Dotted lines indicate variables

Figure 1

As has been explained, the two variables will be set separately in the method presented here. First, C areas are inserted where necessary and possible, based on the available space. Next, all A and H areas are set to zero or some other minimal value. Then the vertical positions of all entities are assigned. If it is possible to fit all entities into the available space, the A and H areas and times are increased to some larger value, where it is possible to do so, and another attempt is made to obtain a fit. This process continues until a fit is impossible, whereupon the previous allocation is used, or until the horizontal dimensions of all entities can be increased no further.

A clear constraint on the amount of A, H, and C area allowable is the length of working storage; each column individually can be no longer than this.

It is harder to arrive at a definition of optimum for these settings because it is greatly dependent on the object system and on the individual programs within the system. For instance, a simulation or system test program will usually be allotted less advance area than a program performing an essential system function.

In every case, however, it is possible to construct a function which yields a value, for each possible setting of advance and holdover area, for each entity, such that an optimum solution results when these values are as nearly equal as possible. The required function can be in the form of a matrix, if no other method is feasible. In some cases maximum and/or minimum desired times might be used in conjunction with the function.

One possible such function, which applies to the systems used as models, represents the amount of probable program operating time contained within advance and holdover areas. If these probable times, (not areas), weighted appropriately for special cases, are as nearly equal as possible, an optimum or near optimum solution should result. An approximate formula for such a function is given:

$$A = w\left(\sum_{c=1}^{n} P_c \, T_c\right)$$

where: A is probable advance time for the entity.

w is a weighting factor for the entity.

c is any one of the n columns in the core rectangle (each representing the operation of a single program) in which an entity has space reserved for it before it is needed by a program.

P_c is the probability that the program in the cth column will operate in the same cycle for which the entity in question is needed. It could be set to zero for all simulation or system test programs if desired.

T_c is the probable operating time of the program in cth column, assuming that the program does operate and the system load is that for which optimization is desired.

The function is admittedly a crude one, except where only one conditional program appears in the advance time for an entity. However, it is quick to compute. It may also be that some other weighting factor than probability would be more useful; if so, this could be supplied to the allocation program in its place. The decision as to what does indeed constitute an optimal setting of advance and holdover times is left open; the function above is included only as one possible answer.

A function similar to that defining advance time will, of course, be used to define holdover time. A definition of communication time is unnecessary, because a communication area between two appearances of an entity either exists completely or not at all.*

It is clear that, aside from hand designated entities and areas of storage, all system entities can be considered to be in working storage. This is not to say that transfers are necessary for all; some entities in working storage may well remain in primary storage throughout the system cycle. One of the goals of the allocation process will be to minimize the number of transfers required. Including all entities in working storage merely allows for a consistent treatment of A, H, and C areas for all entities. If the designation of a permanent

*The question of whether an entity must have the same or can have a different address for different appearances is system and computer dependent. The allocation methods can easily adapt to either case.

storage area is desired, this can be done after the working storage allocation is finished, and will consist only of renaming a storage area, not of moving any entities. Entities which remain in primary store throughout the cycle will be kept together automatically by the working core allocation method.

It should be re-emphasized here that a practical, rather than a rigorous method is aimed for particularly in this part of the problem. A great deal of sophistication in setting A, H, and C areas does not seem warranted unless more stringent standards for the desired outcome can be established. This does not seem likely with a static allocation.

The process of fitting the entities into the available space, once the horizontal dimensions have been decided upon, is described in the next section. For the present, it is necessary to know only that such a process exists, and that if it cannot fit the entities, no direct clues are available to indicate the probable reason (or at least none have occurred to the present writer).

The complete method of working-storage allocation is now presented:

1. Insert communication areas wherever they are necessary, in other words, wherever two nonconsecutive programs have the same non-temporary entity in their environment.
2. Determine the environment for every program with no advance and holdover areas and with the communication areas presently extant. If all environments individually fit into working storage, proceed to 3. If not, remove communication areas* according to the following rules until a fit is obtained:

*If any column is too long after all C areas have been removed, the problem is not one of allocation.

a. Remove first that area which appears in the largest number of columns which require shortening. (The columns which need shortening may change with each removal; a re-appraisal is therefore necessary each time).

b. If a does not determine a unique entity, temporarily change the communication areas involved into advance and holdover areas by removing them only from the columns which require shortening and any columns in between. Then, using A for the advance time and H for the holdover time thus created, compute for each entity involved:

$$\frac{A}{H}(A + H) \text{ or } \frac{H}{A}(A + H), \text{ whichever is smaller.}$$

Choose the entity which has the largest value for the above function and remove that communication area (and restore the others).

3. Attempt to fit all entities into the available space, with no advance and holdover areas. If they do not fit, go back to 2 and remove additional communication areas by using a smaller value for the available space.

4. Increase advance and holdover times as follows:

a. Choose a maximum value for A and H times which is larger than the previous one.

b. Insert, up to the capacity of each column, advance and holdover areas to bring the times up to the maximum desired, if possible. Do this by an iterative process, on each pass increasing areas by

one column only, until a pass is made with no possible increase. If more than one entity can be extended into a column, choose that one which presently has the smallest value for the time being extended. Never extend a time so that it exceeds the maximum value.

5. Attempt again to fit the entities into the available space. If they fit, return to 4. If not, the most recent allocation that did fit is the final one.

A few facts about this process should be pointed out. The reason for removing as few communication areas as possible is that each such removal necessitates two transfers: a WRT and a RDS. The reduction of advance or holdover areas can only necessitate a single transfer. The function given in 2, b is intended to ensure, if possible, that when a communication area is removed, the holdover and advance times for the necessary transfers are both as large as possible.

The process as given will almost immediately produce an allocation which fits, then will improve upon it steadily. This is desirable if the time available for running the allocation is limited; a usable allocation is obtained even if the program is cut off before it is finished. The amount of increase in the maximum value for A and H times for each iteration will have to be determined empirically. In certain situations it might be better to increase the time greatly each time a fit is obtained and decrease it greatly when a fit is impossible using a sort of binary bracketing technique. However, the attempted fitting of the entities will often require more time when a fit is impossible than when one is possible.

Note that the amount of secondary storage available is not a consideration in choosing communication areas to remove. The scheme as outlined minimizes the number of transfers, and therefore the amount of secondary storage, required. If space is not available for all necessary entities, the problem is one of system design, not of allocation. This assumes that secondary storage is not time-shared as is primary storage. The problem of time-sharing secondary storage is not approached here.

4.2 FITTING ENTITIES INTO WORKING STORAGE

Once the horizontal dimensions of entities have been set by whatever method, the hardest part of the allocation job must be tackled: that of fitting the entities into the primary store so that the total amount of space required is as small as possible. A fit satisfying this condition will henceforth be referred to as a solution to this problem.

The method outlined here, of finding a solution, works as follows: successive complete allocations are tried, each an attempted improvement on the previous one. The improvement scheme is designed to reduce drastically the size of the subset of combinations that is tried, without ever eliminating from this subset all solutions. No proof presently exists that the subset tried will indeed contain at least one solution; all that can be said at present is that a solution has been found in every case that has been tried. It should be pointed out that a solution is often not necessary; it is required only that the entities fit into the <u>available</u> space, not the shortest possible space. It should also be pointed out that a solution may sometimes be possible, but impractical of attainment, because of time restrictions.

Determining the minimum possible space required for a given set of entities is by no means a simple problem; indeed, in some cases it cannot be done without actual allocation. It is fairly easy, however, to decide upon the lower limit of this minimum. If the longest column is allocated in the shortest possible way, the total space required cannot be less than this for any column. It is quite often possible to pack the longest column in this way; this then indicates a solution and further trials are pointless.

The working storage allocation algorithm for entities of known dimensions is now presented:

1. Set the addresses of all entities equal to the first address of working storage. Put all entities into Class 1.
2. Allocate the entities one by one as follows:
   a. From among the entities not yet allocated, choose the one with the smallest address. If more than one of these exists, choose the one which appears in the most columns; if more than one such exists, choose the one which has the greatest distance from its first to its last column; if more than one of these exists, choose any one of them. The entity thus chosen will be referred to as Entity A.
   b. Allocate Entity A to its present address. Change the addresses of all entities meeting the following criteria to the address of Entity A plus the length of Entity A. The criteria are:

1) The entity must be as yet unallocated.

2) It must have an address less than or equal to the address of Entity A plus the length of Entity A.

3) It must appear in at least one column in which Entity A also appears, and must not be mutually exclusive with Entity A.

c. In preparation for future improvement of the allocation, set up a parenthetical list of entities for Entity A. Eligibility for this list depends on the class of Entity A, as follows: If Entity A is in

Class 1 - include all entities which meet the criteria listed below.

Class 2 - include all and only those entities which appeared in parenthesis for Entity A in the previous allocation.

Class 3 - include all entities which meet the criteria listed below, and which also appeared previously in parentheses for the entity which Entity A replaced.

Criteria referred to above for an entity to be put in parenthesis for Entity A:

1) Its address must have been changed by the placement of Entity A. (This excludes Entity A, among others).

2) Its address, just before Entity A was allocated, must have been the same as that of Entity A.

3) It must appear in some column in which Entity A does not appear or Entity A must be mutually exclusive with some third entity with which this entity is not mutually exclusive.

3. When 2 has been performed on every entity, a complete allocation has been made. Only one allocation is kept at any one time; this most recent one will be kept if it is the first one, or if it requires less total space than the previous one (i.e., if the largest ending address of any entity is less than the largest ending address of any entity in the previous allocation). "Kept" means that the complete allocation, together with its parenthetical list, will be stored, and any further manipulations will be performed on it only.

   At this point, if an improvement is to be attempted* (on the best allocation chosen above), try to realize one as follows:

   a. Determine which entity has the largest ending address; call this Entity B.

   b. Choose the entity (most recently inserted in a parenthetical list) which has at least one common column with Entity B, or which appears in parentheses for an entity which has at least one common column with Entity B. If no such entity exists, no further improvement is possible, and the allocation is finished. Call the chosen one Entity C. Call the entity in whose parenthetical list Entity C appears, Entity D.

   c. Change Entity C's address to that of Entity D; Entity C thus replaces Entity D. Put Entity C into Class 3, and remove it from Entity D's parenthetical list.

---

*An improvement might not be attempted if:
a) It is clear that a solution has been found, or
b) The entities have been fitted into the available space, or
c) The allotted time has elapsed.

d. Reset the addresses of Entity D, and all entities which were allocated after Entity D, to the first address of working core. Put all these entities into Class 1.

e. Put all entities which were allocated before Entity D into Class 2.

f. Repeat 2 completely (for all entities). Note that the Class 2 and 3 entities will retain their present positions.

4. When an improvement is no longer deemed necessary, the allocation is finished. This could happen either when the entities have been fitted into the smallest possible space, or when they have been fitted into the available space.

A more informal description of the allocation process will doubtless make it easier to grasp. In addition, since no proof exists that the process outlined will always lead to a solution, the only way to support whatever validity it has is to outline some of the reasoning which led to it.

The basic process of allocation on each single pass is to start at the top of working core, locating entities one by one, in each case pushing down all other entities with which an unwanted overlap would occur. Locating the widest one that will fit in every case tends to lead to a picture which looks roughly like Figure 2. However, it can also lead to a situation like the one in Figure 3, which can obviously be improved upon, as shown in Figure 4 or Figure 5.

It seems to the writer that when two entities are located in one of the relationships shown in Figure 6, no improvement is possible. Only when the relationship is one of those shown in Figure 7 does the choice of which entity to locate first depend on what other entities are present. In other words,

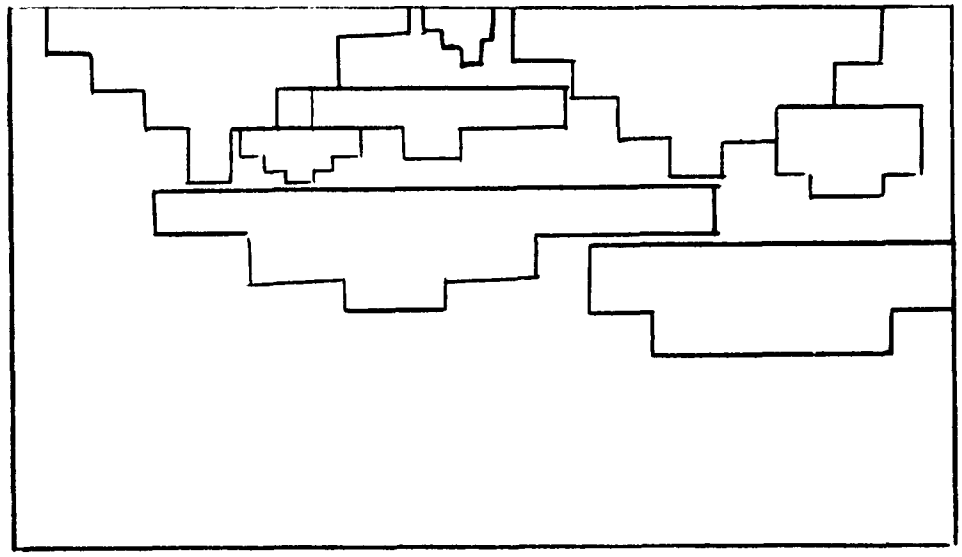


Figure 2

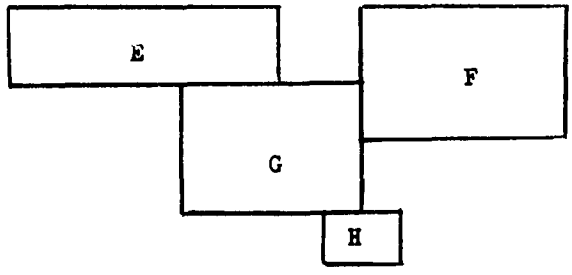


Figure 3

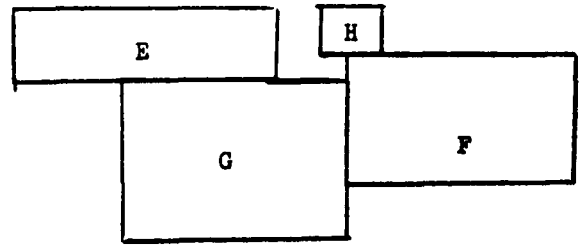


Figure 4

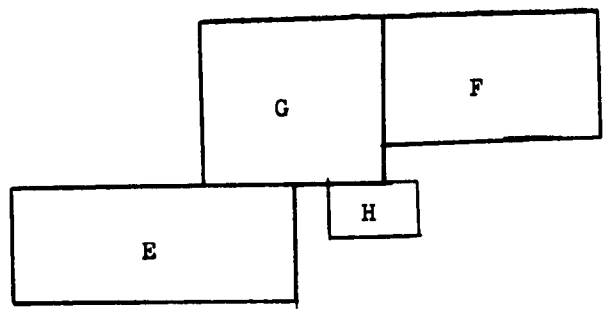


Figure 5

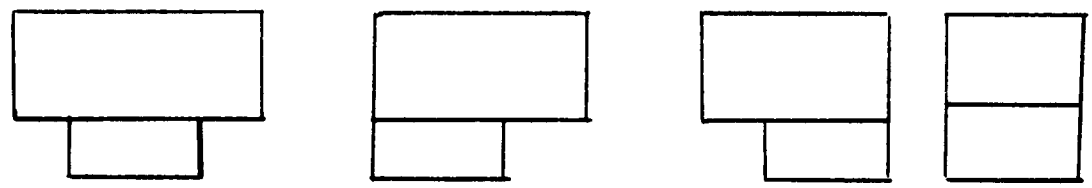


Figure 6

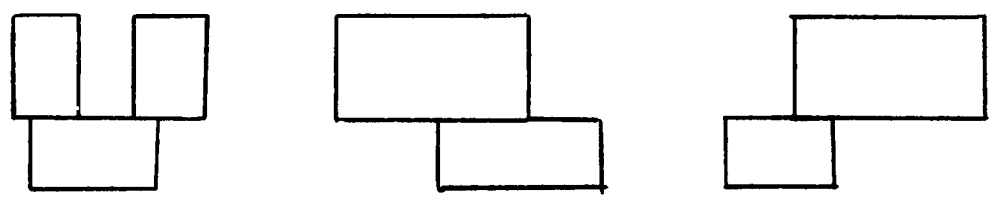


Figure 7

where the first entity completely "covers" the second, the pair cannot exert any adverse influence on the location of other entities, and only one way of allocation need be tried.

The parenthetical lists, then, are made up of all displaced entities which are not completely "covered" by the displacing entity for which they are in parentheses. Because the goal is to pack the longest column if possible, the displaced entities are not included in parentheses unless they could be packed as tightly as the displacing entity. The parenthetical lists are thus intended to be lists of all entities which might profitably be placed in place of the entity for which they are in parentheses.

Each attempted improvement is accomplished simply by changing one of the previous choices as to which entity should be located at a specific point. At least one of the entities involved in the change is required to have a common column with the last entity, because, if the allocation can be improved, the last entity must obviously be located at an earlier address; this can be accomplished only if something in one or more of its columns is moved out of the way. The improvements are always attempted on the best allocation attained thus far; much time is saved by so doing. If the improvement scheme is valid, it is valid for <u>any</u> allocation, including the best so far.

Consider the effect of removing the restrictions described, that is, to include all unallocated entities in parentheses for the entity being allocated, and when attempting an improvement, to do so on the most recently obtained allocation, changing any of the previously made choices without any restrictions on columns. The allocation scheme as outlined, but with these restrictions removed, will cause a trial of all possible orders of allocation.

Note that the maximum number of passes that can be made, before the allocation is either improved upon or finished, is equal to the total number of entries in the initial parenthetical list for the allocation. This number, in turn, has a maximum equal to $n(n - 1)/2$, where n is the number of entities; in practice, because of the restrictions, the actual number of passes made between improvements is a tiny fraction of this maximum. Note also that the improvement sequence must terminate, because on each pass one possible order of allocation is tried and eliminated from further consideration.

Therefore, if the reasons for the restrictions are valid, the method exactly as outlined will find a solution.

The restrictions were arrived at by observing the action of the scheme on actual allocations, then attempting to improve the efficiency of the scheme without impairing its validity. Further refinement may be possible.

4.3 GROUPS AND HAND ALLOCATIONS

Cases may arise in which it is necessary or desirable to locate a set of entities adjacent to each other in working storage; such a set will be called a "group." It may likewise be necessary in some cases to assign an absolute address to an entity, to hand allocate it. Both cases can be handled by the allocation method described, with some modifications. The necessary changes are described separately only for clarity.

Groups may be necessary, either because parts of the same table must be read or written separately but must be together and in order in primary storage, or because it is profitable to read at one time several different entities, which must therefore be adjacent to each other in both primary and

secondary storage. The first of these reasons would necessitate a group of fixed order, the second a group of variable order. At present, it is intended that the decision to group entities will be made by the programmer, and not by the allocation program.

It is useful to divide groups into two classes. A group will be called "normal" when it is possible to arrange the entities in the group in order such that each succeeding member never appears as environment for any program for which its predecessors do not also appear. (A fixed-order group is considered to have only one possible order). All groups not meeting this restriction will be called "unwieldy."

Normal groups fit in very neatly with the logic of the allocation method, and present only a small problem. Unwieldy groups, and individual hand allocations, on the other hand, may present the problem of optimally filling a section of working storage, while at the same time achieving an optimum allocation of the entire area. The problem of optimally filling the space between members of an unwieldy group, as shown by the arrow in Figure 8, is in itself complex enough so that the cost of solving it would be inordinately large. The problem is therefore not approached here; it appears far more practical to put on the user the burden of wisely grouping and hand allocating entities. A method will be outlined, however, whereby the allocation process can be made to respect, if not to optimize in every case, hand allocations and unwieldy groups.

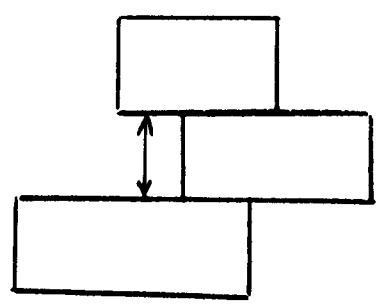


Figure 8

One restriction on groups is included here: no entity can be a member of more than one group. This rule eliminates the necessity for elaborate legality and relationship checks; the allocator is not intended to be a list processor or tree-structure analyzer.

The only effect that groups and hand allocations have on the setting of advance, holdover, and communication areas is that unwieldy groups should be avoided by selective setting of these areas, if possible; this is a simple matter. The following additions and changes to the placement scheme will allow it to handle all groups and hand allocations:

1. Allocate all hand allocated entities first, and include them on a "protected entity" list. Do not change either these allocations or this part of the protected list in any future allocation pass.
2. When choosing an entity to allocate, choose the eligible one that has the smallest address. An entity is eligible if it has not yet been allocated and also satisfies any one of the following conditions:
   a. It does not belong to a group.
   b. It is the first (that one occupying the most columns) of a normal group.
   c. It belongs to an unwieldy group.
3. Before allocating a chosen entity which is not a group member, make sure that it does not overlap with an entity on the protected list. If it does, change its address to the address plus the length of the protected entity and rechoose an entity to allocate. If no overlap occurs with any protected entity, allocate the chosen entity at its present position.

4. Before allocating a chosen entity which is a group member, proceed as follows:
   a. Collect all the members of the group and arrange them in order, according to address if the group is variable order, according to the given order if the group is fixed order.
   b. Increase the addresses, if necessary, of any members of the group, so that the address plus the length of each member is greater than or equal to the address of at least one other member of the group (in other words, pull the group together). If it is necessary to change any addresses in this way, rechoose an entity to allocate.
   c. If no addresses were changed, make a trial allocation of the group in its present position. Increase the addresses of any group members necessary, so that there is no overlap between members.
   d. If any member of the group, as placed in the trial allocation, overlaps with any protected entity, change its address to the address plus the length of the protected entity, and rechoose an eligible entity to allocate. If no such overlap occurs, allocate the entire group to its present (trial) position.
   e. If the group placed is of the unwieldy type, put all of its members on the protected entity list.
5. In setting up parenthetical lists, add the following rules:
   a. Do not set up a parenthetical list for any group member except the first of a normal group.

b. Include, in the parenthetical list for the first of a normal group, all entities that would normally be eligible for inclusion in parentheses for any member of the group, if a were not followed.

c. Never include a group member, other than the first of a normal group, in any parenthetical list.

6. Before each attempted improvement pass, remove from the protected list all entities except hand allocations.

The purpose of these additions and changes is of course to cause groups to be treated as single entities, and to protect hand-allocated entities. Variable-order groups are allowed to fall into place in the order in which they fit tightest with other entities.

It should be mentioned that the grouping of entities in normal groups will usually have no adverse effect on the allocation, and will not interfere with the search for an optimum allocation. Unwieldy groups and hand allocations, however, unless wisely chosen, may adversely affect both these things. It will be noted that some improvement in the packing of entities around unwieldy groups and hand allocations may be achieved by the normal improvement scheme; however, no attempt is made to go further than this.

5. DRUM ALLOCATION

A simple method of drum allocation has been developed for the systems used as models, and is presented here. It should be applicable to any system with the following characteristics:

1. Drums are used as a major secondary store.
2. The chief problem to be solved by the allocation is that of fitting all necessary entities on the available drums in such a way that as few drum accesses as possible will be needed to transfer any single entity.
3. The sequence and timing of drum RDS is unpredictable enough so that minimizing drum latency time by locating entities optimally is impossible, except as in 2.

A nominal drum field length of 2048 registers is assumed for the purpose of illustration. It is also assumed that the total length of entities needing drum assignment does not exceed the drum space available; if it does, the problem is not one of allocation.

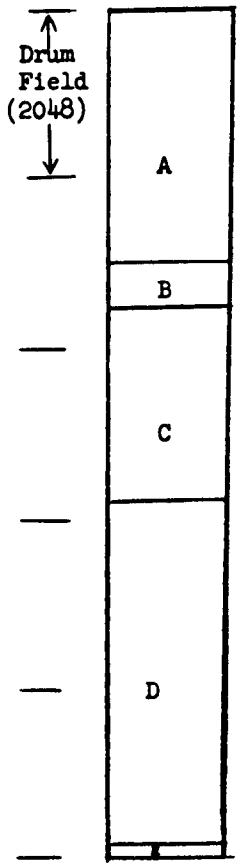
It follows from 2 that as few entities as possible should be unnecessarily split across drum fields. In other words, a 3000-word entity should appear on two and only two drum fields, if possible.

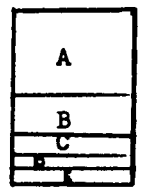
Each entity may be thought of as occupying a certain number of drum fields plus some excess number of registers, in other words, modulo 2048. Thus, a 5000-word entity consists of two full durm fields and 904 registers on a third field. It is obvious that if this entity has a starting address less than or equal to 1144_{10} (2048 - 904) on one drum field, and is continued on consecutive drum fields, it will appear on the minimum three fields. In other words, if the entity modulo 2048 will fit in the available space on one drum field, the entire entity will occupy the minimum number of fields.

The basic allocation method is now presented:

1. Determine the modulo 2048 length ($length_m$) of all entities and arrange them in order, based on this length. Immediately, assign all entities that have a zero $length_m$ to entire drum fields.
2. Assign the unassigned entity with the greatest $length_m$ starting at the beginning of the first free drum field. This entity may or may not occupy several drum fields; in any case, the available space on the last drum field it occupies will be 2048 minus the $length_m$ of the entity.
3. Choose the unassigned entity with the greatest $length_m$, whose $length_m$ is less than or equal to the available space. Assign this entity starting immediately following the most recently assigned entity. The available space will now be the previous available space minus the $length_m$ of this entity.
4. Repeat 3, until there is no entity whose $length_m$ is less than or equal to the available space. At this point a string has been completed. Start at the beginning of the next drum field and assign the next string in the same way. Keep assigning strings until all entities are assigned or no unassigned drum fields remain.

It may clarify matters to think of all entities as having an actual length equal to their $length_m$, so that each string occupies a single drum field. The two strings in Figure 9 are logically equivalent, for purposes of drum allocation (ignoring number of drums available).





	Lengths	Lengths$_m$
A:	3048	1000
B:	500	500
C:	2300	252
D:	4246	150
E:	146	146
Totals:	10240	2048

Figure 9

It is obvious that the above scheme does not necessarily produce the best possible allocation; that is, it will not always fit all the necessary entities on drums. The drum-allocation problem, too, is combinatorial in nature. No clues pointing to possible ways of improvement such as exist in the working-storage allocation seem to be available in drum allocation. In general, it is manifestly not feasible to try all possible ways of allocation. All that can be done, therefore, is to include some improvement scheme which maximizes the probability of indeed finding an improved allocation. A few as yet untested suggestions for such a scheme are advanced.

Some improvement could probably be produced either by trying all combinations until no more than a specified minimum number of blank registers remain at the end of each string, as the strings are initially allocated, or by trying certain predetermined subset of combinations over the entire allocation, and choosing the one which fits the most entities without unnecessary splitting. It is probably best, when trying combinations, to change the selection of the earlier entities in each string, as this will tend to produce more of a change in the string as a whole. It will probably accomplish more to choose a new entity that differs noticeably in length from the previous selection when changing a selection. Whether or not an improvement scheme is warranted at all will, of course, depend on how tightly the object system fills the available drums.

The method as it stands is expected to produce good results, especially in a large system with a great number of entities of varied length. The reason for picking the longest entity that will fit in all cases is that

this will tend to save the shorter entities until last (when they will be most needed for filler) because fewer unassigned entities remain. This is easily seen if the outcome of assigning the shorter entities first is considered.

It will be noted that the method as described will cause no unnecessary splitting of entities across drum fields. If all drums are assigned, and unassigned entities remain, it is sometimes possible to split these across drum fields wherever there are spare registers. Note that if two adjacent strings have spare registers, the second string can be moved down so that the spares are on consecutive drum fields. Note also that the order of entities in the strings (once they have been picked) matters not a whit, so that exchanging within the string can be done so that the least important entities are split across drum fields. (Figure 1 makes this immediately clear.)

Only hand-allocated entities remain to be mentioned. If these exist, they must be assigned to drums first and other entities assigned around them. If an entity is chosen for assignment that will conflict with a hand-allocated entity, it is necessary only to choose the next entity in the list that does not, if there are any such. A string must end before each hand-allocated entity and a new string must be started immediately after it, if there is any available space.

6. IMPLEMENTATION AND PRELIMINARY TESTING

Certain portions of the allocation methods described have been programmed for the SAGE computer (AN/FSQ-7), with the SAGE application primarily in mind.

Other portions are presently being written and checked out. A certain amount of testing has been accomplished; the results of this may be of interest.

A brief description of the allocation system, which is presently called ALLOCATOR, is necessary. Because the present SAGE control program and sequence parameters are different from the ones presented, the advance and holdover areas cannot be set for SAGE as outlined. In fact, these areas cannot profitably be set by a program at all until a change is made in the present control program. Therefore, it has been decided to require a programmer to construct the sequence parameter table (there is only one in SAGE at present); this table rigidly fixes advance and holdover areas. A program has been written to analyze this table, make legality checks for misplaced transfers, columns of excess length, etc., and determine from it what the advance and holdover areas are. The fitting of entities into core is then done as described.

A program to set advance and holdover areas and construct sequence parameters for a system with a control program similar to the one outlined is being written. It is expected that this will make ALLOCATOR usable for the new BUIC system. This effectively demonstrates both the flexibility of the allocation methods and the advantages of splitting the working-storage-allocation problem into two parts.

The following programs are presently included in ALLOCATOR:

CTR - system control program. Reads inputs, operates other programs, etc.

SPZ - sequence parameter analyzer program. Described above.

ALC - working-core-allocator program. Fits entities into available space.

APR - ALLOCATOR-print program. Puts out documentation, including a two-dimensional working-core picture,* on the direct or delayed printer.

CPL - compool-construction program. Makes legality checks and outputs a binary and symbolic compool.

PRM - permanent-core allocator.

DRM - drum allocator.

Under construction are the following:

ADV - program which sets advance and holdover areas and constructs sequence parameters.

ITM - a simple item allocator. (Based on the same basic design as the drum allocator.)

A very small amount of testing has so far been accomplished; even so, the results so far are quite encouraging. ALC has operated quite fast, even in situations where it was impossible to pack the longest column. Most of the tests so far have been run on a working storage environment of approximately 70-80 entities, 35-45 program operations, with approximately 1/2 to 2/3 of the

*The picture is illustrated in an appendix.

total available core area filled. In these tests, ALC has required approximately one second for its first pass, and about 1/3 to 1/2 second for each complete iteration thereafter. Its longest total time so far has been 192 seconds in a situation where it was necessary to try all combinations (within the restrictions previously outlined).

In contrast, running ALC on the same environment with only one restriction removed (that of column "cover") has yielded operating times of 15 or more minutes, in some cases without producing a solution.

No tests have yet been run on any of the other parts of the allocator.

7. CONCLUSION

The chief concern of this paper has been the solution of the working-storage-allocation problem. A basic approach to the problem has been presented, together with programmable and practical solution methods for its various facets and for closely connected problems such as that of system control. There may be many improvements possible in the techniques outlined. Nevertheless, the methods are usable, at least for some systems, as they stand; where they are not, they can serve as a useful starting point.

It should be pointed out that, although the methods presented were designed to function best in conjunction with each other, they are not necessarily interdependent. For instance, the dynamic-control program, or the drum-allocation scheme, can be used independently, if desired. Even the working-storage-allocation scheme, although it can be used to fullest advantage only with the dynamic-control program presented, can be used with

other control methods. In point of fact, the presently implemented version of the allocator does just that.

It should be clear by now that the entire allocation of a large-scale system revolves around the working-storage allocation. The methods outlined here can be used to build a system-generating system, which would fit any large-scale system into a computer with minimum human effort. It would seem that all such systems have enough in common to make it possible to write one allocator system for all, in which minor adjustments for individual systems could be made. This conjecture could be proven only by further study.

It would also seem but a short step from the methods outlined here to a completely item-based system where the allocation began at the item level. Items would be placed in tables according to their content and use, and the tables would then be allocated as outlined here. At present, items are usually assigned to tables by humans directly. Here again is a subject for a future study.

One problem not approached here at all is that of changing allocations and compools with minimum effort. It should be possible for an allocator to not only fit a system into the machine initially but also to determine the cost of proposed changes and to make these changes with minimum cost. These seem, however, two completely different problems. Much work remains to be done in this area.

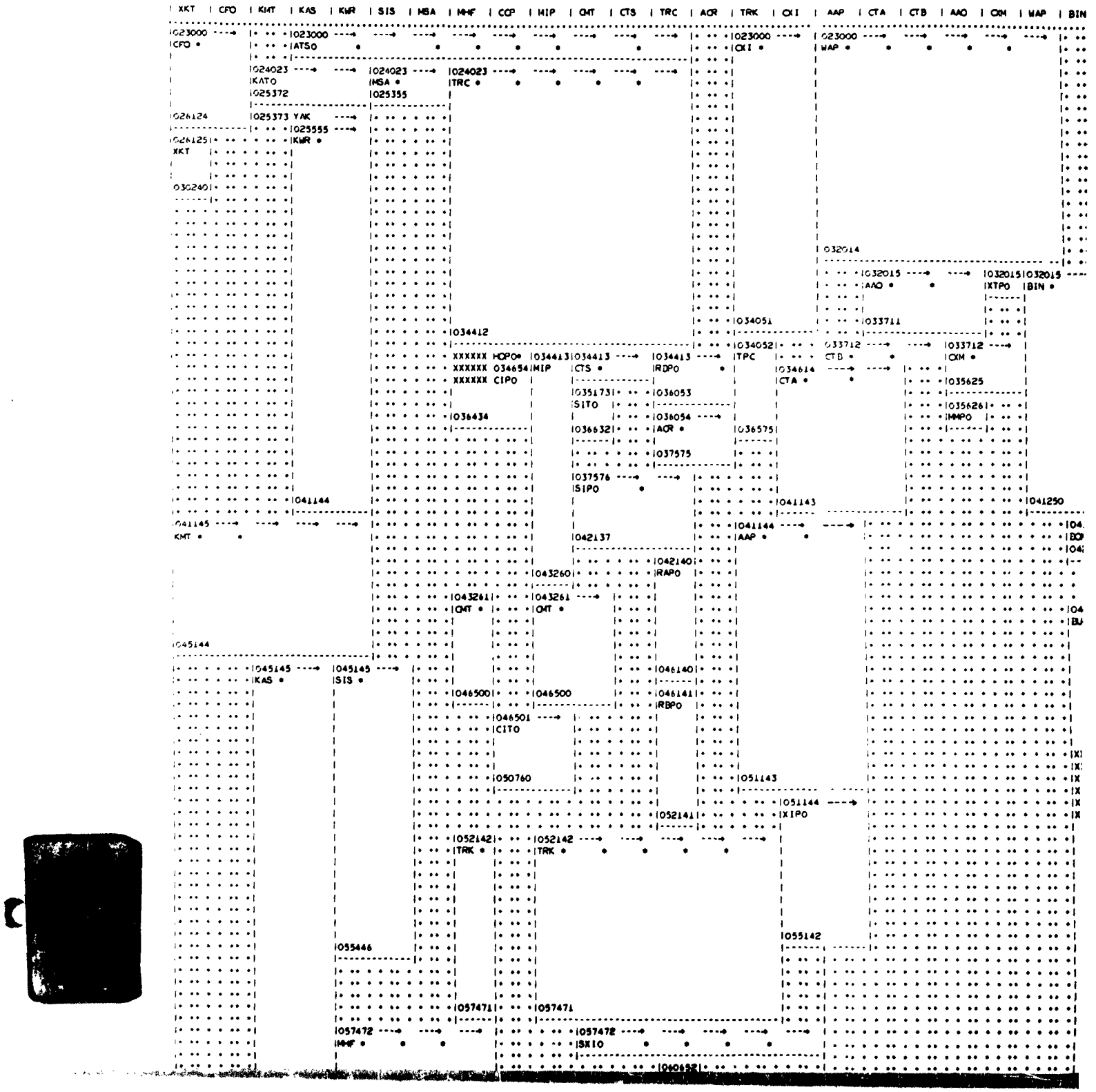
Nevertheless, the methods presented here make possible the construction of a very powerful allocation tool. With such a tool, organizing or changing a large-scale system would be a far simpler and safer task than it is at present.

(Page 42 blank)

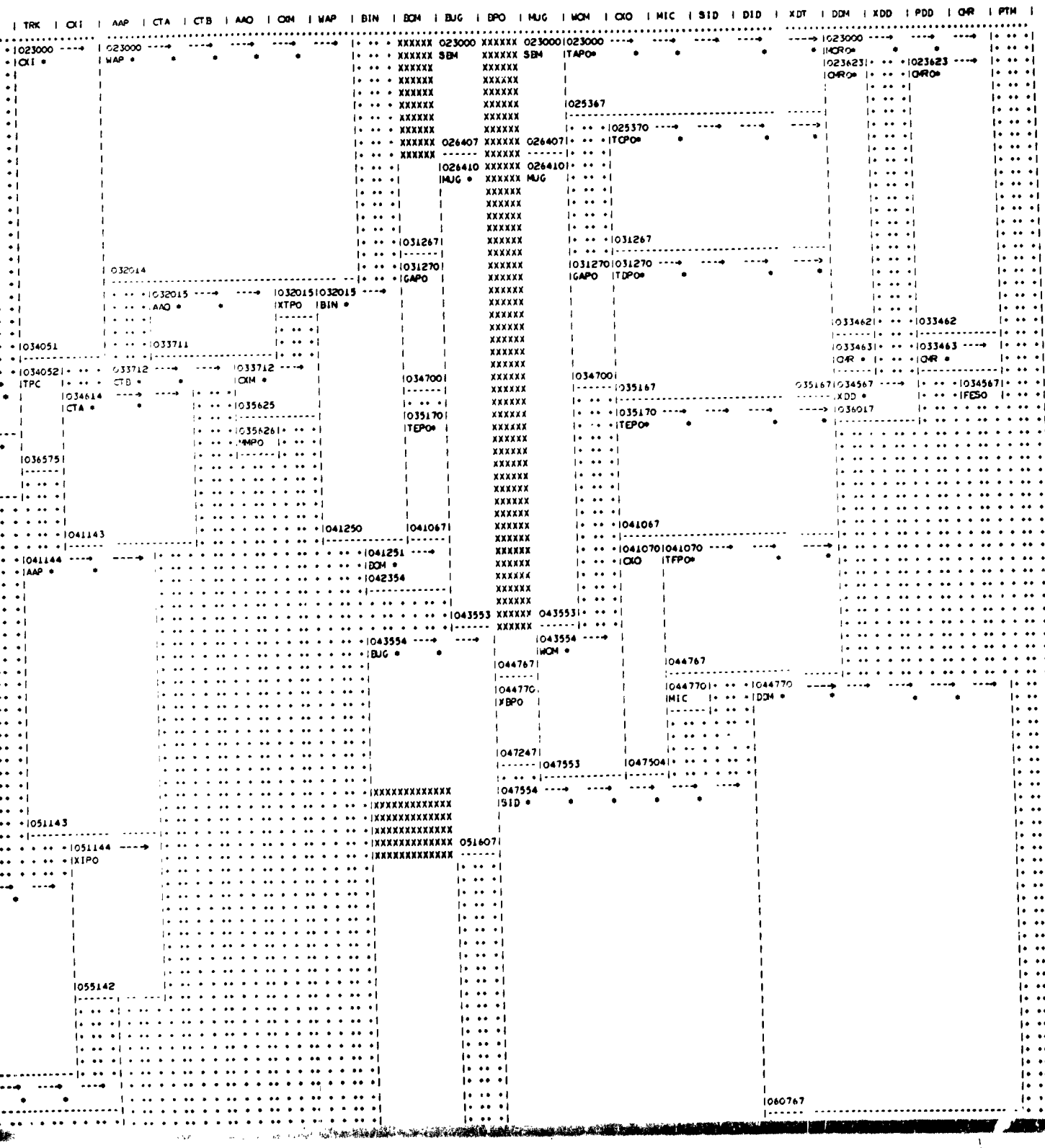
APPENDIX

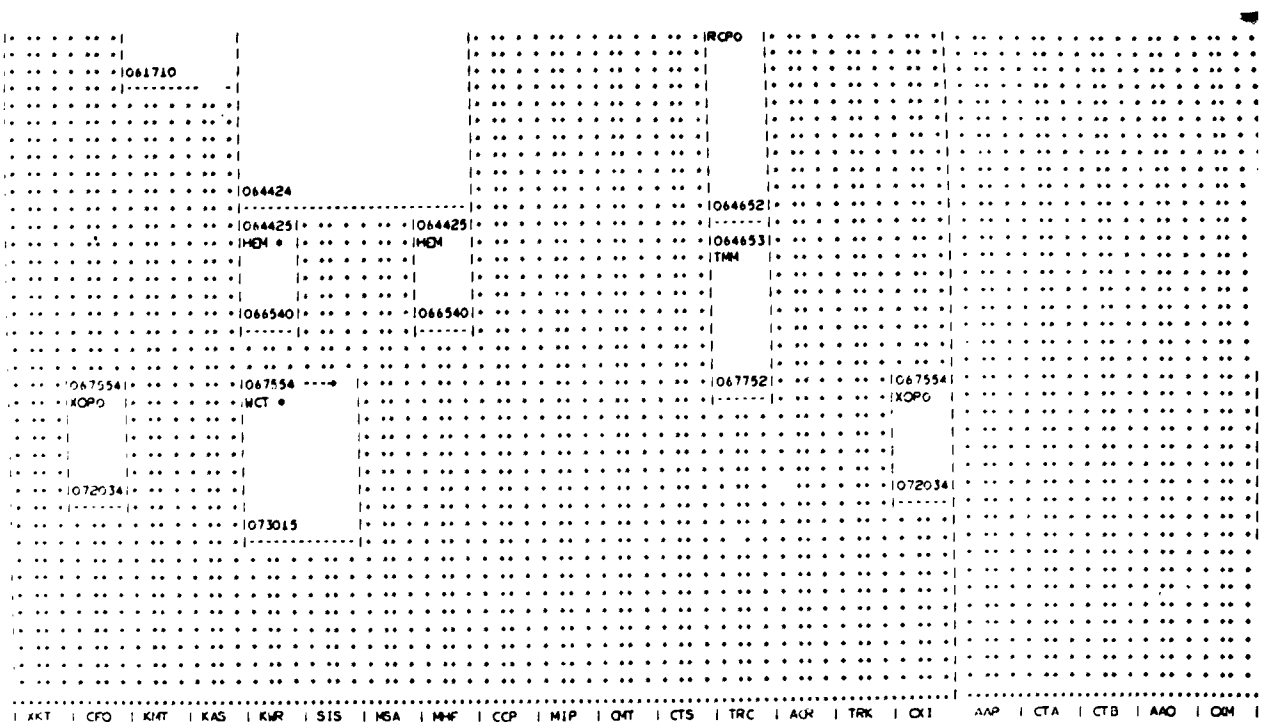
The computer-produced picture on page 43 shows the working-core section of the SAGE system (Model 9). ALLOCATOR was used to determine program environments, from the program listings and the sequence parameter table, and to allocate working core.

In this particular situation, the allocation is less than optimum because of the existence of a (fixed-order) unwieldy group: TAPØ, TCPØ, TDPØ, TEPØ, and TFPØ. Reversing two transfers to make the group normal has enabled ALLOCATOR to produce an allocation more than 700 registers shorter.


XKT
CFO
KMT
KAS
KWR
SIS
MSA
MHF
CCP
MIP
CMT
CTS
TRC
ACR
TRK
CXI
AAP
CTA
CTB
AAO
CXM
WAP
BIN
023000
ATSO
024023
KATO
025372
MSA
025355
TRC
026124
025373 YAK
025555
KWR
026125
XKT
030240
034412
XXXXXX HOPO
XXXXXX 034654
XXXXXX CIPO
034413
MIP
034413
CTS
034413
RDPO
036434
036053
035173
SITO
036054
036632
ACR
036575
037575
037576
SIPO
034051
034052
TPC
034614
CTA
033711
033712
CTB
CXM
032014
032015
AAO
XTPO
BIN
035625
035626
MMPO
041144
041143
041250
041145
KMT
042137
AAP
042140
RAPO
043260
043261
CMT
045144
045145
KAS
SIS
046140
046500
046141
RBPO
046501
CITO
050760
051143
051144
XIPO
052141
052142
TRK
055142
055446
057471
057472
MHF
SXIO
060632

042/102/00


TRK
CKI
AAP
CTA
CTB
AAO
CKM
WAP
BIN
BOM
BUG
BPO
MUG
MOM
CKO
MIC
SID
DID
XDT
DDM
XDD
PDD
CMR
PTM
023000
CKI
WAP
SBM
TAPO
023623
CMRO
025367
025370
TCPO
026407
026410
MUG
031267
031270
GAPO
TDPO
032014
032015
XTPO
BIN
AAO
033711
033712
CKM
CTB
033462
033463
CMR
034051
034052
TPC
034567
XDD
FESO
034614
CTA
034700
035167
035170
TEPO
035625
035626
MPO
036017
036575
041143
041144
AAP
041250
041251
BOM
041067
041070
CKO
TFPO
042354
043553
043554
BUG
WOM
044767
044770
MIC
DDM
044770
BPO
047247
047553
047504
047554
SID
051143
051144
XIPO
051607
055142
060767



XKT	CFO	KMT	KAS	KMR	SIS	MSA	MHF	CCP	MIP	CMT	CTS	TRC	ACR	TRK	CXI	AAP	CTA	CTB	AAO	COM

TOTAL NUMBERS OF SPARES-

17223	17122	12586	6175	3064	12219	18207	7996	14875	7967	9619	12099	3069	16941	8482	7110	9538	14724	16988	16105	17062

SPARE AREAS-

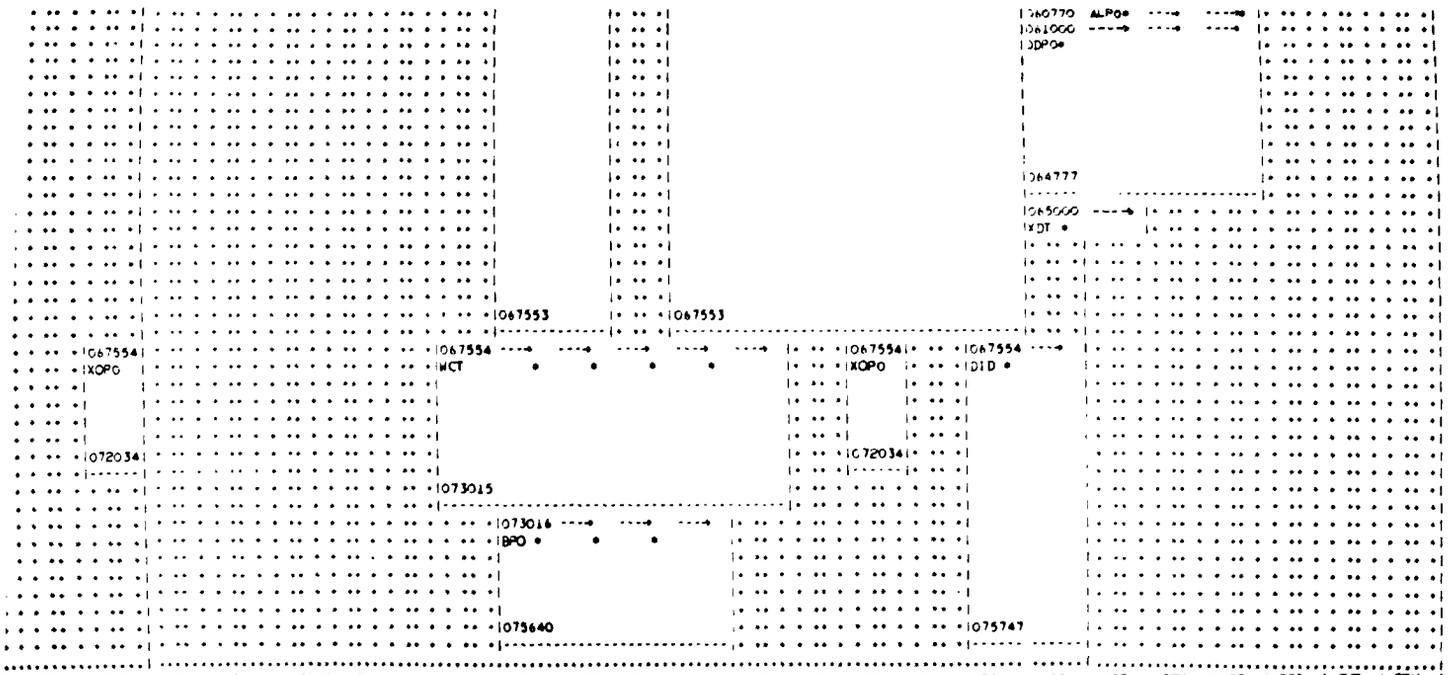
030241	028125	023000	061711	055447	025356	025356	036435	036435	050761	036633	035173	067753	023000	036576	034052	032015	041144	034614	036377	032566
041144	041144	024022	075747	057471	045144	057471	043260	046500	052141	037575	037575	075747	034412	041143	034613	033711	075747	075747	075747	033711
045145	045145	025555		066541	055447	064425	046501	050761	057472	042140	042140		037576	051144	055143	055143				035626
075747	067553	041144		067553	057471	075747	052141	075747	075747	043260	052141		052141	052141	057471	075747				075747
	072035	061711		073016	064425		066541			046501	060652		060652	060652	060652					
	075747	075747		075747	067553		075747			052141	075747		075747	075747	067553					
					073016					060652					072035					
					075747					075747					075747					

ENTITIES INCLUDED IN OVERLAPS-

034413
CTS
035172

034413
HOPO
034653

034654
CIPO
036434



R	TRK	CXI	AAP	CTA	CTB	AAO	CXM	WAP	BIN	BOM	BUG	BPO	MUG	WQM	CXQ	MIC	SID	DID	XDT	DDM	XDD	PDD	CMR	PTM
941	8482	7110	9538	14724	16988	16105	17062	12957	4307	1006	7211	267	1498	8679	2034	4002	1396	1131	4 27	8144	12724	10865	10765	21992

3000 036576 034052 032015 041144 034614 036377 032566 041251 023000 034701 051610 047250 073016 025370 047505 046106 044770 065401 065401 036020 023623 034567 034733 023000
4412 041143 034613 033711 075747 075747 075747 033711 067553 032014 035167 067553 047553 075747 031267 047553 047553 047553 067553 075747 044767 034566 044767 044767 075747

7576 051144 055143 055143 035626 073016 042355 041070 075641 075641 034701 072035 067554 065000 036020 060770 060770
2141 052141 057471 075747 075747 075747 043553 041250 075747 075747 043553 075747 075747 075747 044767 075747 075747

0652 060652 060652 075641 042355 067554 065000
75747 075747 067553 075747 043553 075747 075747

072035 075641
075747 075747

043554 023000 023000
BUG TAPO TAPO
051607 025367 025367

047554 023000 023000
SID SDM SDM
067553 026407 026407

025370 025370
TCPO TCPO
031267 031267

043554 026410
BUG MUG
051607 043553

047554 031270
SID TDPO
067553 035167

035170
TEPO
041067

041070
TFPO
044767

UNCLASSIFIED

System Development Corporation,
Santa Monica, California
CONTROL OF AN AUTOMATIC ALLOCATION
FOR LARGE-SCALE CYCLYING SYSTEMS.
Scientific rept., TM-1042/102/00,
by C. P. Earnest, 15 March 1963, 43p.
Unclassified report

DESCRIPTORS Programming (Computers).

Presents a programmable method of storage allocation, along with a scheme for system control, suitable for large-scale systems where something is known

UNCLASSIFIED

---

in advance about the sequence of program operations, and time of system operation is an important factor. Reports that the allocation is static, because dynamic allocation in systems of this type usually cost more time than it saves. States that the primary store is assumed to be of the continuous, random-access type, such as core memory. Also states that the SAGE and SATIN systems have been used as models in developing this method.

UNCLASSIFIED

UNCLASSIFIED